On 3rd October 2013, High Integrity C++, one of the most respected, longest established and widely adopted C++ coding standards, marked its 10th anniversary with the publication of major new update – Version 4.0. Over the past decade a staggering 24,000 copies of this coding standard have been downloaded.

This whitepaper presents genealogy and philosophy of the latest version of HIC++, and its unique contribution over the previously available coding standards for C++. C++11 changes the game considerably: secret handshakes can be replaced with explicit new syntax. The new HIC++ specifically favors these C++11 features to simplify the rule set and its enforcement. In a similar vein, additional concepts are introduced to further simplify this coding standard. This has created room for brand new rules covering the new features in C++11, e.g. lambdas, rvalue references, and the concurrency libraries. The result is a reduced number of rules, which are more generic, comprehensive, and amenable to automated enforcement compared to the previous version of HIC++. To demonstrate benefits of this new approach, the effectiveness of the new and old versions of HIC++ is compared empirically.
1 Introduction

The original High Integrity C++ coding standard, published on 3 October 2003, pooled best practice advice available at the time [Stroustrup][Effective C++] [More Effective C++] [Effective STL] [Industrial Strength C++] [Exceptional C++], as well as internal know-how at Programming Research. The result was a set of 202 rules and guidelines for the use of the C++ language. Since stylistic issues were largely ignored, and left for the individual organizations to define, this collection of rules and guidelines established a subset of the original programming language [ISO C++ 2003], applicable to any development, including high integrity.

In the intervening decade other publicly available language subsets emerged [JSF AV C++] [MISRA C++], which specifically focus on use of C++ for safety critical applications. In 2011 major changes to the C++ language have been ratified [ISO C++ 2011], which invalidate some of the existing advice, as well as expose gaps relating to the use of new constructs, such as lambdas, rvalue references and new facilities of the standard library. In order to prevent HIC++ from losing its relevance, a major revision has emerged to address these shortcomings.

2 Rule Composition

The relationship between Version 3.3 and Version 4.0 of HIC++ is summarized in Figure 1, and further explained in subsequent sections.

2.1 Retired Rules

A large number of rules in a language subset typically gives rise to rule overlap, and can make enforcement problematic, especially if manual enforcement is used. With this in mind, in version 4.0 of HIC++ [HIC++4] we have opted not to carry forward 80 rules and guidelines from version 3.x [HIC++3]. Some of them are not relevant to the guiding principles of HIC++: code maintenance, portability, readability and robustness. A good
example is the retired guideline 3.1.7, which is simply too subjective, and without a proven case for or against following it:

| retired | Do not use the 'inline' keyword for member functions, inline functions by defining them in the class body. |

Rules that are not susceptible to automated enforcement, thus limiting their practical value, have also been omitted, for example rule 3.3.7:

| retired | Only define virtual functions in a base class if the behavior will always be valid default behavior for derived classes. |

Finally, a small number of rules have become completely obsolete with the new C++ standard [ISO C++ 2011], because they have become constraint violations (compilation errors), or well defined, as is now the case for guideline 10.18:

| retired | Guard the modulus operation to ensure that both arguments are non-negative. |

Of the remaining rules, 20 have been unchanged and 28 have been reworded to clarify enforcement and improve consistency. Other rules have undergone a more extensive reformulation, as detailed below.

### 2.2 Generalized Rules

It is common for a language subset to contain a rule or a guideline banning a particular language construct, e.g. ‘minimize the use of casts’, and another constraining its use, e.g. ‘avoid using pointer or reference casts’. This introduces a level of redundancy, because either the more restrictive rule should be followed, or the more lenient one. In our view it is more practical to have a single rule – typically the more restrictive one – and use the rule deviation mechanism, to permit localized non-compliance. With this in mind we have merged several rules from HIC++ V3.3, to remove such a two-tiered approach, and avoid rule overlap.

Rules which have similar rationale, but pertain to different features, represent another aspect of redundancy in HIC++ V3.3. A good example of this situation is the following pair of rules:

- Rule 8.4.2 (V3.3) Declare each variable on a separate line in a separate declaration statement...
- Rule 8.4.7 (V3.3) Declare one type name only in each typedef declaration.

We have realized that the key consideration is their common rationale, and decided to merge these rules into a single rule*, with a similar approach applied to analogous cases of duplication.

In some cases, by adopting a modern coding concept we were able to side step a complete set of rules concerned with a problematic C++ feature. For example, the following new rule allows pointers and resource handles to be properly managed, addressing exception safety, and preventing memory leaks and dangling pointers, previously covered with separate rules:

| 3.4.3 (V4.0) | Use RAII for resources |

Overall, by following these principles we managed to replace 44 overlapping rules with 16 more generic ones, as detailed in an Appendix of the HIC++ manual [HIC++4].

### 2.3 C++11 Updates

In the latest version of the C++ standard [ISO C++ 2011], new syntax has been added to simplify the definition of special member functions, namely =delete and =default, rendering obsolete rules in HIC++ V3.3 and other language subsets relating to the declaration of copy constructors, copy assignment operators and destructors:

| Rule 3.1.3 (V3.3) | Declare or define a copy constructor, a copy assignment operator and a destructor for classes which manage resources. |
| Guideline 3.1.13 (V3.3) | Verify that all classes provide a minimal standard interface against a checklist comprising: a default constructor; a copy constructor; a copy assignment operator and a destructor. |

* HIC++ V4.0 Rule 7.1.1: Declare each identifier on a separate line in a separate declaration
These rules have been replaced in HIC++ 4.0 with one with clearer text and much simplified enforcement:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5.1</td>
<td>Define explicitly =default or =delete implicit special member functions of concrete classes</td>
</tr>
</tbody>
</table>

Similarly, special identifiers final and override allow simplification of rules relating to overridden virtual member functions. Additionally, the new standard library header <cstdint> defines† size specific type aliases, e.g. int32_t, which are preferable to similar type definitions in user code, as advocated by previous best practice‡. In all, 15§ rules have been updated based on introduction of better alternatives in C++11.

### 2.4 Other Significant Modifications

In a few cases, namely 12 rules, we felt that extending the rules, by making them in effect more restrictive, would benefit their justification, and enforceability. As an example:

<table>
<thead>
<tr>
<th>Rule 9.2 (V3.3)</th>
<th>Only throw objects of class type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1.1 (V4.0)</td>
<td>Only use instances of std::exception for exceptions</td>
</tr>
</tbody>
</table>

Conversely, 12** other rules were relaxed, as we believed them to be overly restrictive, without a good case matching the guiding principles of HIC++, as detailed in Section 2.1. As an example:

<table>
<thead>
<tr>
<th>Rule 11.3 (V3.3)</th>
<th>Specify the name of each function parameter in both the function declaration and the function definition. Use the same names in the function declaration and definition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.1 (V4.0)</td>
<td>Make parameter names absent or identical in all declarations</td>
</tr>
</tbody>
</table>

### 3 New Rules

Finally, 61 new rules have been added, covering topics detailed below. This brings the total rule count in HIC++ V4.0 to 155, down from 202 in V3.3. Each rule now has a comprehensive explanation with examples of compliance and non-compliance, and is categorized using its relevant clause and sub-clause from the text of the language standard [ISO C++ 2011]. The new categorization ensures that related rules are grouped, and allows for easy navigation within the document, as well as cross referencing with the language standard.

#### 3.1 Prior Language Subsets

In a separate whitepaper [PRQA Overlaps], overlaps between publicly available language subsets [HIC++3][JSF AV C++][MISRA C++] were examined. In particular, it was found that several rules are common to JSF AV C++ and MISRA C++ but absent from HIC++ V3.3. Clearly, these represented potential gaps in the language subset. After review, 15 rules have been incorporated into the new version of HIC++, which are heavily influenced by their JSF AV C++ (and also MISRA C++) counterparts. One of the rules has also been updated to fit the new language specification, as elaborated in Section 2.3. In addition, 6 rules unique to MISRA C++ have also been incorporated, with one updated to match C++11.

Notably, we have borrowed from JSF AV C++ the notion of an interface class, and defined it in HIC++ as follows:

- all public functions are pure virtual functions or getters, and
- there are no public or protected data members, and
- it contains at most one private data member of integral or enumerated type

Addition of this concept simplifies formulation of a rule limiting use of multiple inheritance:

† if available in a particular compiler implementation
‡ HIC++ V3.3 Rule 8.4.6, JSF AV C++ Rule 209, MISRA C++ Rule 3-9-2
§ also 4 have been merged as detailed in Section 2.2
** also 5 have been merged as detailed in Section 2.2
10.3.1 (V4.0) Ensure that a derived class has at most one base class which is not an interface class

This brings the new version of HIC++ partially closer to both JSF++ and MISRA C++, as the overlap increased to from 76†† to 91 rules in common. However, at the same time the HIC++ unique area has been extended by 40 rules, as detailed in the subsequent sections. Figure 2 presents a side-by-side comparison of JSF++ and MISRA C++ overlaps with HIC++ V3.3 and V4.0 respectively. In order to represent the comparison accurately, the effects of rule merges (as described in Section 2.2) and rule deletions (see Section 2.1) are ignored. As can be seen from Figure 2, HIC++ V4.0 (combined with V3.3) now accounts for 55% of the combined rule set, compared to 46% previously. The remaining rules are beyond the scope of HIC++ as detailed in Sections 2.1 and 5.

![Figure 2: Overlaps between HIC++, JSF AV C++ and MISRA C++. HIC++ slices are denoted with red hue, JSF++ with yellow, and MISRA C++ with checkered pattern.](image)

CERT C++ Secure Coding Standard [CERT C++] is another commonly used coding standard. As this is the most recent entrant to the C++ coding standards arena, it shares many rules with the prior coding standards. A considerable proportion of the remaining rules are not concerned with the use of C++ language features, e.g. rules constraining use of the POSIX API or focused on code behavior. For this reason it is not strictly speaking a subset of the C++ language. Only two CERT C++ rules are quoted in HIC++ V4.0, with the remaining rules being redundant (as discussed in Section 2.2), previously covered elsewhere, or not directly related to C++.

3.2 Previously Unpublished

PRQA is actively involved in various C++ forums [WG21][ACCU], where on occasion C++ vulnerabilities are discussed. Through such unpublished sources, we have identified 4 additional rules not specific to C++11 for inclusion in the new version of HIC++. These rules can be identified in the HIC++ Manual [HIC++4] by the absence of any external references.

3.3 C++11

The new version of C++ standard [ISO C++ 2011] has revised the language definition substantially, and in particular has added several new features, e.g. lambdas and rvalue references. Mainstream C++ compilers have already implemented these features, so it is paramount to address dangerous aspects of their use. C++ gurus have started disseminating guidance on the use of these features [GOTW] [Effective C++ ’11], and we have based 7 new rules on this material. In addition, we have added 17 rules originating from unpublished

†† 72 HIC++ v3.3 rules are in common with MISRA C++
sources, as described in Section 3.2, and our independent analysis of the text of the language standard [ISO C++ 2011].

3.4 Concurrency

Previously, ISO C++ lacked any support for multithreading and synchronization. This has been rectified in the latest version of the standard [ISO C++ 2011], by explicitly providing data race guarantees, and extending the standard library with the provision of the Atomic Operations and Thread Support libraries. C++ and concurrent programming gurus have embraced these features and started to provide guidance on their use [Sutter Hardware][Williams Concurrency]. Based on these sources we have formulated 11 rules, and also 1 from unpublished sources, specifically relating to concurrency facilities in C++11. Notably we have provided an additional class high_integrity::thread, which wraps and enforces correct usage of std::thread, and provided its definition in the following rule:

\[ 18.2.1 \text{(V4.0) Use high integrity::thread in place of std::thread} \]

4 Enforcement in Practice

So far we have provided justification for all the rule changes and additions in HIC++V4.0. However, an open question remains as to how these modifications affect the enforceability of HIC++. As HIC++ is one of the most popular language subsets for C++ [VDC], it is important to understand how code written for ISO C++ 2003, which adheres to HIC++ V3.3, complies with regards to HIC++ V4.0. If there is no a major increase in non-compliance, projects can switch to HIC++ V4.0 with relative ease, and safely start using and migrating to new C++11 features. In order to verify this assumption we compared compliance of several open source projects to HIC++ V3.3 and V4.0.

We analyzed each codebase with QA-C++ (PRQA’s Static Analysis tool) and the HIC++ Compliance Module V3.3, and then suppressed all non-compliance warnings, i.e. deviated from all specific instances of rule violations. We then analyzed the code with QA-C++ and the HIC++ Compliance Module V4.0, and examined the new compliance warnings. Table 1 lists all the projects and their corresponding deterioration in compliance, in column 3. As can be seen from the table, for projects that already comply with HIC++ V3.3, the additional burden in fully\(^\dagger\) migrating to the latest version is modest, at 15% of the original effort of compliance on average.

\[^{\dagger}\] This could be performed in a piecemeal fashion, with the rules that create the most non-compliance tackled last, alternatively by deviating from these specific rules.
Table 1: Comparison of violations unique to HIC++ V4.0 (3rd column) and V3.3 (4th column). The averages were calculated by treating the whole collection of projects as a single codebase.

To complete this study we inverted the HIC++ versions, and reanalyzed the codebases, as detailed above. This has allowed us to see if the retired rules (as described in Section 2.1), the overlapping rules (as per Section 2.2), and the relaxed rules (as per Section 2.4) are causing unwanted violations with V3.3, which are now absent with V4.0. Column 4 of Table 1 demonstrates that HIC++ V4.0 avoids 17% of unnecessary violations on average. Therefore, on balance the new version of HIC++ requires less effort to enforce, while constituting a more comprehensive language subset, as detailed in Sections 2 and 3.

5 Comparison with the MISRA Guidelines

In March 2013, the latest version of the MISRA C guidelines was published [MISRA C:2012]. This is the third edition of the guidelines which cover the use of C in critical systems. Questions may arise as to the difference in philosophy of HIC++ V4.0 and MISRA, as well as the absence of the explicit MISRA C:2012 meta concepts, specifically:

- Rules versus Directives,
- Decidable versus Undecidable, and
- Analysis Scope

The key difference between the HIC++ and MISRA (also JSF AV C++) standards is the target application domains. The HIC++ rule set has been selected such that all rules are appropriate and can be applied to any domain, resulting in improved code quality, without adding overhead due to overly restrictive rules. In the case where critical or safety related issues are to be considered, then HIC++ should be used as a companion coding standard with a more restrictive rule set.

5.1 Difference in Philosophy

Many of the rules selected for HIC++ share roots with rules in MISRA. The philosophies of the rule sets are therefore very similar, however, HIC++ differs slightly from MISRA in terms of the target domain. MISRA C:2012 targets software for use in critical systems. In such environments the cost of failure, no matter how rare, is significant and so rules must cover as many failure cases as possible. For example, MISRA C:2012 Dir 4.12 disallows the use of dynamic memory. This is appropriate within a critical embedded environment where an attempt to acquire more memory than is available would result in the failure of the system. HIC++ does not have such a restriction, as many applications are designed for running on hardware with gigabytes of memory. Similarly, MISRA C:2012 Rule 21.6 covers the functionality provided by the I/O libraries. The use of I/O in critical applications is disallowed because its behavior can be unspecified, implementation defined or undefined. However, I/O is a requirement for many applications in non-critical domains and so such a rule cannot be generally applied.

5.2 Rules vs Directives

MISRA C:2012 explicitly distinguishes between issues that can be detected through analysis of source code alone (“Rules”), and items that are subjective or that require knowledge from an external source (“Directives”). The rule set of HIC++ has been selected and written in such a way that enforcement should be possible through source code analysis. For example, the HIC++ rule relating to assembler usage is worded such that any use of the ‘asm’ declaration is a violation of the rule:

| 7.5.1 | Do not use the asm declaration |

In comparison, MISRA C:2012 Dir 4.2 requires that use of assembly language be documented. It cannot be determined from the source code alone if such documentation exists and is correct.

§§ Corresponding to the retired rules, see Section 2.1.
5.3 Decidable vs Undecidable and Scope

HIC++ does not include the concept of decidability or scope as described by the MISRA C:2012 guidelines, however HIC++ does include the concept of demonstrability. In MISRA C:2012, a rule is deemed to be 'decidable' if in every case, it is theoretically possible for a static analysis tool to determine if code complies to the rule. For example, a rule may be deemed undecidable if code behavior could be dependent on input from an external source. On the other hand, the HIC++ concept of demonstrability requires that the issue is not possible in the source code:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Ensure that data loss does not demonstrably occur in an integral expression</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Ensure that pointer or array access is demonstrably within bounds of a valid object</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero</td>
</tr>
</tbody>
</table>

6 Conclusions

This whitepaper presents genealogy of the latest version of HIC++ as reviewed HIC++ V3.3, new rules and rule revisions specific to C++11, additional rules common to JSF++ and MISRA C++ not already in HIC++, and also concurrency rules. As a key consideration, rules are formulated to avoid conflicts or overlap, in order to improve enforceability. Rule presentation has been improved, with a new categorization based on the relevant clause and sub-clause of the C++ language standard, and a comprehensive justification with examples. The principles of the latest HIC++ and MISRA C guidelines have been discussed and the differences noted, the foremost of which is applicability of HIC++ to any development domain. Finally, it has been shown that the rule changes have beneficial effect on enforceability, with improved signal-to-noise ratio, when compared to the previous version of HIC++.
References


[Exceptional C++] Herb Sutter: Exceptional C++, 2000, Addison-Wesley


About the Authors

Wojciech Basalaj
Wojciech is a senior developer at PRQA, primarily responsible for the Dataflow analysis component of QA·C and QA·C++. Previously, he looked after pre- and post-sales support of PRQA’s products. Through hands on and customer facing experience he amassed knowledge of C++ coding standards, and is a co-author of High Integrity C++.

Richard Corden
Richard is a lead developer at PRQA, primarily responsible for the enhancement of QA·C++, a static analysis tool for the C++ language. He has extensive experience in the development and implementation of C++ coding standards, and is a co-author of MISRA C++ and High Integrity C++. He represents PRQA at the ISO C++ Committee [WG21] meetings.

About PRQA
Established in 1985, PRQA, ISO 9001 and TickiT certified, is recognized throughout the industry as a pioneer in static analysis, championing automated coding standard inspection and defect detection, delivering its expertise through industry-leading software inspection and standards enforcement technology used by over 3,000 companies globally.

PRQA’s industry-leading tools, QA·C, QA·C++ and QA-Verify, offer the closest possible examination of C and C++ code. All contain powerful, proprietary parsing engines combined with deep accurate dataflow which deliver high fidelity language analysis and comprehension. They identify problems caused by language usage that is dangerous, overly complex, non-portable or difficult to maintain. Plus, they provide a mechanism for coding standard enforcement.

PRQA has corporate offices in UK, USA, India, Ireland and Netherlands, complemented by a worldwide distribution network. Find out more at www.programmingresearch.com

Contact Us
PRQA has offices globally and offers worldwide customer support. Visit our website to find details of your local representative.

Email: info@programmingresearch.com
Web: www.programmingresearch.com and www.codingstandard.com

All products or brand names are trademarks or registered trademarks of their respective holders.